Scalability and Availability in EPIWORK

Recent problems

In the last year, the support infrastructure for the Epidemic Marketplace web platform suffered several issues that severely affected its availability. We took the only immediate solution, migrating the platform to a newly built datacenter that would allow us to improve the platform's stability and, through it, its availability. Performing this migration made us realize that we had to design a way to scale the platform in order to prevent another outage from happening. Scaling the platform had to take into account the EPIWORK specifications and requirements, so we had to design a way to make the platform scalable globe-wide instead of just inside a single datacenter.

The first part of the report focuses on what is an EM node, its architecture and component relations, the second part focuses on how to make an EM node scalable to guarantee availability and fault tolerance, and the third part explains how to expand the EM platform by adding new EM nodes, independently of its geographical location.

Node EM structure

As it stands, an EM node is a group of 7 services hosted by 6 independent virtual machines running CentOS, all in one physical server running CentOS and XEN Hypervisor for virtualization.

The services are Drupal, LDAP, Fedora Commons, FedoraGSearch, Apache Solr, Web services and Sendmail, and each one has its own assigned virtual machine (besides Apache Solr and FedoraGSearch, which are in the same one).

Drupal is a framework for developing websites whose components are a set of code files and a database. Besides these two components, we require an Apache HTTP server to allow web access to the website. The database is MySQL.

We use OpenLDAP as our LDAP application, along with an Apache HTTP server to host it.

Fedora Commons is the platform's resource repository, whose components are a web app, a database and a file system. The resources are stored in the designated file system location, while the MySQL database stores information required by the repository. Fedora Commons requires a servlet container to host it, for which we decided to use Apache Tomcat.

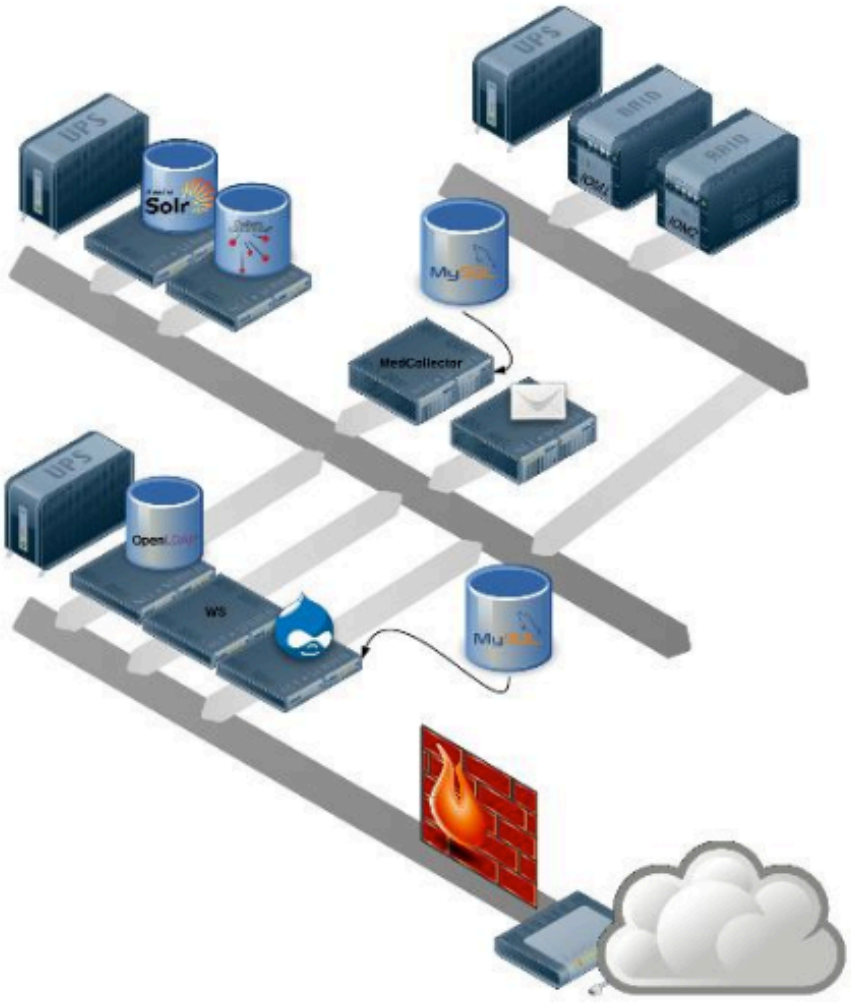
FedoraGSearch (Fedora Generic Search Service) is part of the Fedora Service, and performs indexing of Fedora FOXML records (resource contents). To be noticed that this service is a plugin of Lucene and Zebra, so it has to be implemented on top of an application of this kind.

Apache Solr is a Lucene search engine that requires a servlet container to host it, and like with Fedora Commons, we decided to use Apache Tomcat. Apache Solr is the search engine to which FedoraGSearch was added to allow the search in Fedora Commons. No additional components are required for it to work.

The web services are a set of python executables, hosted by an Apache HTTP server with a Python WSGI module.

Sendmail is our mail service, consisting in the Sendmail application for CentOS along with Lighttpd to host it. We chose Lighttpd over Apache HTTP Server due to the Lighttpd having the best performance.

Picture 1 presents a diagram of how a node looks like in terms of virtual machines, services hosted by them and disposition across different virtualization lines (we consider a virtualization line a set of IP range with specific properties).



Pic. 1: Aspect of a node EM across virtualization lines.

Guaranteeing node availability

There are two possible types of fails: virtual machine failure and service component failure. Adapting the service components so they can be used across several virtual machines solves both scenarios. Each replicated service is configured in a master-slave scenario, where write requests are always forwarded to the master service, and read requests are balanced between all replicas. Due to write requests only being

forwarded to the master, whenever the master fails, write requests are not executed. This is done to improve consistency by never leaving the master in an inconsistent state.

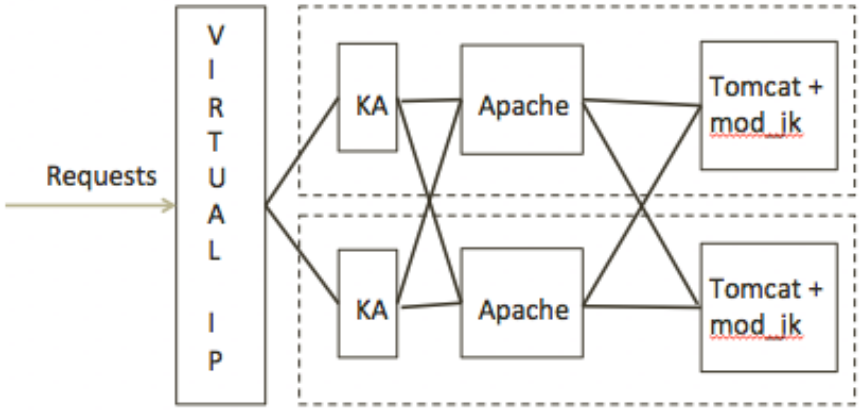
We use a load distributor that defines a virtual IP, so that a call to that IP will be forwarded to one of the virtual machines that host the service, so the request can be routed to an active instance of the service. Three configurations for load distributing were considered: HAProxy, Apache HTTP Server + Heartbeat and Apache HTTP Server + Keepalived. HAProxy didn't have a module to implement load balancing across tomcat instances, so it was discarded. Using Keepalived turned out to be much simpler than heartbeat while having the same properties, so it was the chosen solution. To support Keepalived, we had to add an Apache HTTP Server between a service and Keepalived, guaranteeing load balancing to both Apache HTTP Servers and the service.

Keepalived natively supports replication, but it's necessary to define a master and slave instances. Whenever the master instance is running, requests are forwarded by it to the apache instance running on its virtual machine.

Apache HTTP Server has no dynamic behavior in regards to replicated instances, and it's only necessary to replicate configuration changes.

Approaches to services that need additional configurations/applications in order to support replication are discussed below.

Picture 2 presents a diagram of how a tomcat-based service environment would look if it had two separate virtual machines, each one hosting tomcat, apache and Keepalived. In normal operation, any request would reach the network through a single virtual IP, and the master instance of Keepalived will send it to the master instance of apache, which will then send it to the master instance of tomcat.



Pic. 2: Diagram of a replicated Tomcat-based service.

Fail scenarios:

- Keepalived: If at least one instance is running (whether it's a master or slave), the service is kept available. Every request will be managed by Keepalived, which will send it to the master apache, and so on.

- Apache HTTP Server: If at least one instance (master or slave) is running, Keepalived will forward the request to the apache instance, which in turn will deliver it to the master tomcat.
- Service: If at least one service instance is running, the request will reach and it'll be delivered to the web app running under tomcat environment.

We conclude that if at least one instance of each component is running, the request will be delivered to the web app and a reply will be sent to the client. Even if each running component is hosted by a different virtual machine, the request will be correctly routed, triggering a response.

- Fedora Commons

Replicating Fedora Commons requires assuring that replicas are all updated when any operation is made. The native solution for Fedora Commons is journaling. A journal is a file to which the master Fedora Commons instance writes resource updates. Each replica keeps its own journal and, whenever the master receives a write operation (whether it's a new resource, a resource update, etc.), it executes the operation and writes the update in every replica's journal. The master will close the journal whenever it reaches its maximum lifetime or size, after which the replica will read the journal, execute the updates and archive the journal. The next journal operation executed by the master will trigger the creation of a new journal.

- Drupal

We use Keepalived to have a single virtual IP to which requests are made, an apache instance for load balancing and web hosting, and the Drupal framework. Drupal is a set of code files and a database, which in our case, is a MySQL database. In order to have a replicated Drupal, it's necessary that both the code files and database stay updated across replicas. Keeping the code files consistent is achieved using an SVN repository (which is already in use), while keeping the MySQL databases consistent is achieved using the Master-Master feature of MySQL.

- LDAP

We use Keepalived to have a single virtual IP to which requests are made, an apache instance for load balancing and web hosting, and the LDAP service. LDAP works on top of a MySQL database, so using the MySQL Master-Master feature allows replicated work.

- Webservices

Since the Webservices are a set of static code, no additional configurations or implementations besides Keepalived and apache to ensure a replicated environment.

- FedoraGSearch

FedoraGSearch requires no additional configurations or implementations besides Keepalived and apache to ensure a replicated environment.

- Apache Solr

Replicating Apache Solr instances is done using Apache Solr Master-Master feature which ensures the code and index changes are replicated throughout all replicas.

- Sendmail

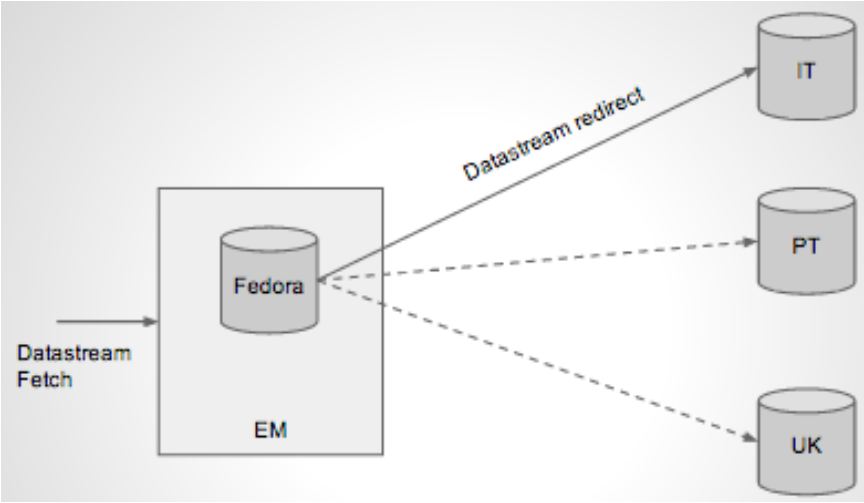
The Sendmail has to run under Lighttpd instead of apache due to compatibility issues, which means Keepalived can't be used without any adaptation. Nevertheless, Lighttpd has a module named FastCGI (already implemented), which allows a load balancing configuration without the need for Keepalived.

Connecting multiple EM nodes

In this section we cover how to allow several EM nodes to co-exist in different geographical locations, while dealing with the privacy requirements of EPIWORK. The project requires that the data producers are responsible for storing and managing their resources, meaning that only people to whom they give access permissions can see the resource's content. We present three approaches to solve the problem, and the reasons to choose the one we did.

The first approach consists in having a centralized Fedora Commons repository while having the actual data stored in other EM nodes. This is done by storing all resource's metadata inside the repository and having the metadata referencing either an object inside the repository, or an external object stored in another EM node. Instead of a decentralized solution, we get a centralized repository with a distributed file system.

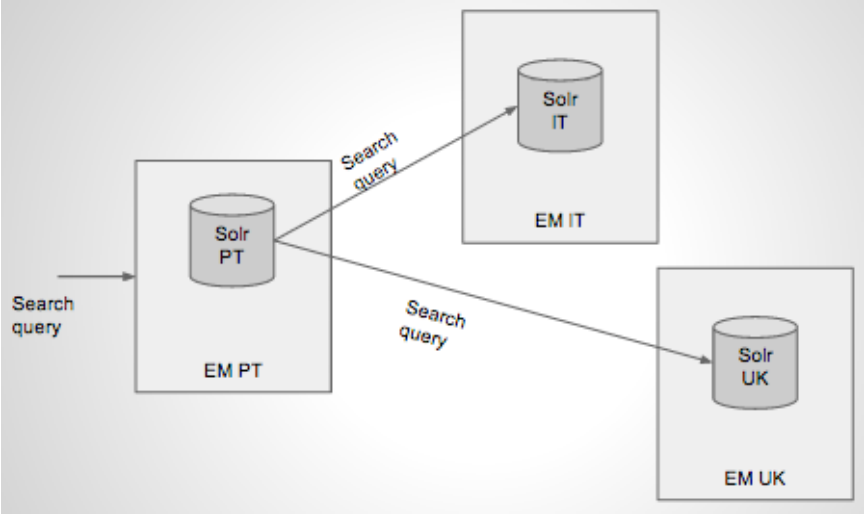
This solution requires less maintenance costs due to only requiring a single repository, but it doesn't guarantee the required level of privacy and resource control to the owners.



Pic. 3: Centralized fedora with distributed file system.

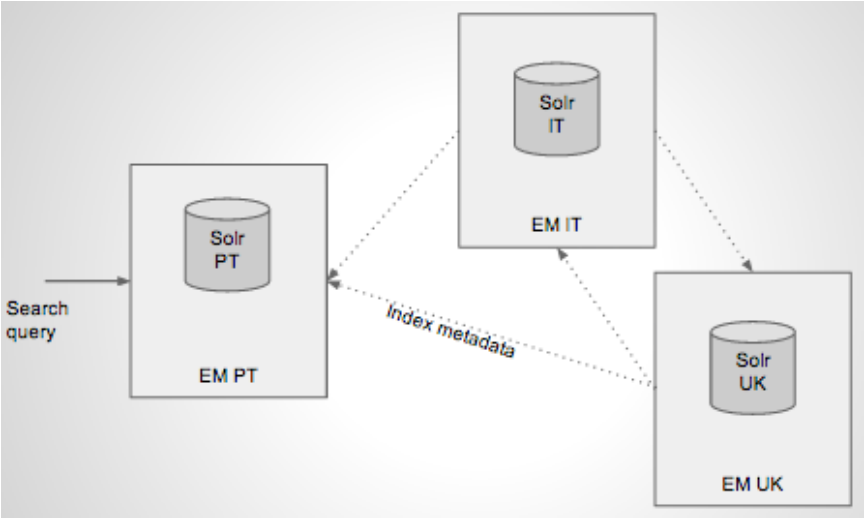
The second solution consists in having multiple Fedora Commons instances, each one with their own resources (that may or may not be replicated), and uses the Apache Solr servers to search for a resource across several nodes. Each node's Apache Solr instance only has the indexes for resources inside the node's repository, and no information is maintained about other resources. Picture 3 shows the model of responses: any query received by an Apache Solr server is forwarded to the desired servers, which will return the results.

This approach satisfies the privacy requirements since only the node's owner has control over the node's resources, while guaranteeing that requests for that resource's metadata coming from any EM node will always have success. The disadvantages are the performance overhead of communicating with external nodes in order to get a resource. If a query is made to several servers, the query will take a considerate amount of time to be solved. Although the Solr servers would need to be exposed so that requests could be made to them, but the risks of such scenario are mitigated by using a VPN.



Pic. 4: Apache Solr servers across three countries resolve a query.

The third approach consists in having at least one repository in each node, being that only that node's owner has control over the resources. The difference to the second approach is that each Apache Solr instances will periodically send to every other instance the additions to its index. A query for a resource wouldn't be propagated to other nodes, but there's a possibility of returning false negatives to a query. There is a window of time between an Apache Solr server updating its indexed and informing other servers about the new resource, during which a query made to an Apache Solr server that hasn't got updated will return no results, even though the resource exists in the system. Although this approach is decentralized (which requires additional maintenance costs) and has the referred false negative period, we still feel this is the right solution to apply, due to introducing no performance overhead on queries while still satisfying the privacy requirements. By not affecting query performance, user interaction with the platform isn't affected.



Pic. 5: Replication of Solr server indexing